Title: Design Churn as Predictor of Vulnerabilities?

Aram Hovsepyan, Riccardo Scandariato, Wouter Joosen
iMinds-DistriNet, KU Leuven
3001 Leuven Belgium
first.last@cs.kuleuven.be

Maximilian Steff
Free University of Bozen-Bolzano
Bolzano, Italy
first.last@unibz.it

Abstract

This paper evaluates a metric suite to predict vulnerable Java classes based on how much the design of an application has changed over time. We refer to this concept as design churn in analogy with code churn. Based on a validation on 10 Android applications, we show that several design churn metrics are in fact significantly associated with vulnerabilities. When used to build a prediction model, the metrics yield an average precision of 0.71 and an average recall of 0.27.

Keywords: Security vulnerability prediction; machine learning; software metrics; Android applications

Title: Design Churn as Predictor of Vulnerabilities?

Introduction

Security vulnerabilities are a serious threat to any organization as an exploit can cause severe monetary and reputation damage. It is essential to detect and mitigate software vulnerabilities before the software product is released. Verification and validation activities, such as security testing and code review are effective means in reducing the number of post-release vulnerabilities. However, such quality assurance is not only inexpensive, but it is also best done by engineers specifically trained in software security (McGraw, 2006). Hence, tools and techniques that can help identify components that are more likely to contain vulnerabilities can provide substantial support to the security engineers who can focus their attention and efforts on higher risk components.

One of the possible approaches to predict vulnerable components is to build statistical models using software metrics. Historically, prediction models based on software metrics are known to be very effective in defect prediction (e.g., Basili, 1996; Menzies, 2007; Nagappan, 2005). Since recently, various studies have investigated the effectiveness of vulnerability prediction models based on software metrics. As opposed to defect prediction, vulnerability prediction is much more complicated as vulnerabilities are typically few in number. Nonetheless, various studies have demonstrated the effectiveness of vulnerability prediction models based on mutually complementary set of software metrics. A number of works has investigated the predictive power of *implementation-level* code measures, such as size and complexity (Shin, 2011; Chowdhury, 2011). *Design-level* measures, such as coupling, dependencies between components, were observed to be efficient especially in terms of recall (Zimmermann, 2010; Shin, 2011). The afore-mentioned measures are *static* in the sense that they consider a software system at a specific point in time. Recent works have shifted their focus towards *evolutionary* measures, such as code churn, which is a measure of the amount of code changed within a software unit over time. The evolutionary measures could provide an even higher performance than *static* measures (Shin, 2011). However, there is a clear lack of research with respect to the use of evolutionary design-level measures in the domain of vulnerability prediction.

This paper focuses on evolutionary design-level measures. The contribution of this paper is an exploratory study of whether the changes to the dependency structure of a software system could be used as predictors of vulnerable software components. We consider a software system as a graph where nodes represent classes and directed edges represent dependencies between the classes. We then compare the changes to the dependency graphs across different versions of a software system. We refer to this metric as design churn in analogy with code churn. Our previous work has shown that design churn metrics are excellent predictors of defects (Steff, 2011). We now apply design churn metrics to the domain of security and perform a validation on ten Android applications. In this paper, we show a statistically significant association between design churn and security vulnerabilities. We also build a vulnerability prediction model based on design churn that provides good performance in terms of precision (0.71 on average), but low recall values (0.27 on average).

The rest of the paper is organized as follows. In section 2, we present the overall research methodology including the application selection, the goals of our investigation and the exact experiment setup. In sections 3 and 4, we present and discuss the results and describe the most important threats to validity. In section 5, we describe the related work. Finally, section 6 presents the conclusions and provides an overview of the future work.

Research Methodology

In the context of our study, we experiment on ten Android applications, where each application is represented by a set of Java classes. Each class is then regarded as *vulnerable* if there is at least one vulnerability in the code, as reported by a static code analyzer. Otherwise the class is considered as *clean*.

The overall goal of this study is twofold. First, we investigate whether there is an association between design churn metrics and vulnerable classes. Second, we build a vulnerability prediction model based on design churn metrics and evaluate its predictive power. To encourage the replication of this study, all the experimental materials are available online, including the data we have used, description of the techniques we have used to extract the presented metrics, etc. (KULeuven, 2013).

Design Churn (Independent Variable)

Each application release can be represented as a directed graph where classes represent the nodes, while dependencies between these classes represent the edges. We define design churn as changes to these graphs across different releases of an application. Before identifying and measuring structural change, first we have to define and extract the structure itself from Java source code. Note that in the context of this work we do not consider the AndroidManifest.xml file although it could contain essential information relevant to vulnerabilities. We partially compile the Java source code and we extract the dependencies from the resulting byte-code. Dagenais and Hendren (Dagenais, 2008) presented PPA, a tool for compiling and analyzing partial Java programs harnessing the Eclipse JDT. We tapped into the traversal of the produced Abstract Syntax Trees (AST) and retrieved all inheritance, usage and delegation relationships between classes and methods. We retrieved all dependency relationships not only to Java files within the respective application version, but also to Android libraries. Vulnerabilities may arise from ill-conceived calls to system functions. This is why we have decided to include these technically external dependencies.

We have selected two different types of class-level metrics in order to measure design churn. First, we leverage the number of added and deleted in- and outgoing dependencies for each class. As mentioned earlier, for each application release we build a dependency network with classes as nodes and dependencies as directed edges. Such network provides us with static fan-in and fan-out measures for each class. Dependencies have a source and a destination classes and are labeled by the point of origin and destination in each class. For example, if method a in class A calls method b in class B, the resulting dependency label is the string “A.a#calls#B.b”. For each pair of classes in each pair of consecutive releases of a system we compare the set of dependency labels to determine the number of added and deleted dependencies between them. Note that we omit a class from measurement when it has been added or deleted in either of the two releases. Second, we use the neighborhood hash graph kernel (NHGK) introduced by (Hido, 2009). NHGK first produces bit labels for each node in the graph. Then it hashes the bit labels of each node’s neighbors along with outgoing dependencies into its label. Hence, for each node in the graph NHGK encodes the neighborhood information of that node into its label. If the label of a node (i.e., class) does not change across different releases of an application that means that the neighborhood of that node has not changed. We use the NHGK labeling and hashing three times. Bit labels for iterations 1, 2 and 3 reflect the neighborhood information of each node, of the node’s neighbors, and the node’s neighbors’ neighbors, respectively. These labels allow us now

to compare the neighborhood structure of nodes across releases. The number of distinct bit labels (NDBL) measure introduced in (Steff, 2011) reflects changes for each neighborhood size. However, we use it only pairwise on bit labels l for consecutive releases r-1 and r, such that we obtain for each class:

$$NDBL_t = |sgn(l_r - l_{r-1})|$$

for iterations t=1, 2, 3, which is either 0 (no change) or 1 (change) due to taking the absolute value of the sign function (sgn).

Table 1 provides an overview of the design churn metrics we have used. Note that we ignore code refactoring from our measurement. If a method name changes all method calls from and to this method will change. The same applies to class names. The rationale behind this is that we cannot easily match methods and classes across releases with high confidence. Hence, we opt to treat all such changes equally and consider related dependencies to have changed between releases.

Table 1
[Class-level design churn metrics]

Metric	**Type**	**Description**
added fan-in	count	new dependencies towards a class
deleted fan-in	count	removed dependencies towards a class
added fan-out	count	new dependencies from a class
deleted fan-out	count	removed dependencies from a class
$NDBL_1$	boolean	changed neighborhood of a class
$NDBL_2$	boolean	changed neighborhood of a class's neighbors
$NDBL_3$	boolean	changed neighborhood of a class's neighbors' neighbors

Vulnerable Classes Identified by a Tool (Dependent Variable)

Despite their popularity, very few vulnerabilities in the context of Android apps have been published in vulnerability databases. Vulnerability databases like the National Vulnerability Database (NVD) typically report only a few vulnerabilities. Historical data – which is necessary for our analysis – is either missing or insufficient. This is why we have used Fortify's Source Code Analyzer (SCA), an automated static code analysis tool with built-in support for Android apps, to identify the vulnerabilities in the selected apps (and for all considered releases). While static code analyzers are known to produce false positives (Austin, 2011) the use of SCA is an objective and repeatable technique that enables replication of this study. As mentioned before, a Java class is considered as vulnerable if Fortify reports at least one vulnerability warning for that class. Otherwise, the class is considered as clean. For the purposes of our analysis we have used version 5.10.2 with Fortify Secure Coding Rules version 2012-1.

Association

The first goal of our study is to investigate whether there is an association between each of the design churn metrics in Table 1 and the dependent variable. We formulate the first research question (RQ) as follows:

RQ1: Are design churn metrics associated with vulnerabilities? We leverage a number of statistical tests in order to investigate this research question. Added/deleted fan-in and fan-out measures all belong to the interval scale. Hence, we used the Welch's t-test in order to

investigate whether there is a statistically significant location shift between the means of these metrics for vulnerable and clean classes.
The NDBL measure (independent variable, or treatment) takes values in the categories 0 and 1, i.e., the variable belongs to the nominal scale. The same holds for the vulnerability status (dependent variable, or outcome). We use the Chi-square test in order to determine whether the independent and the dependent variables are statistically independent. If not independent, there is an association between the two variables and we measure the strength and direction of the association by means of the relative risk. Relative risk (RR) is frequently used in the analysis of binary outcomes, like in our case. If RR is lower than 1, the probability that a Java class is vulnerable is greater if its NDBL is equal to 1. If RR is greater than 1, the probability that a Java class is vulnerable is greater if its NDBL is equal to 0.

We have devised two different setups to investigate RQ1. First (RQ1.1), we have created a dataset that consists of all Java classes from one release of each application. The rationale behind this decision is that the mix of all applications is representative for the broader domain of Android applications. Per application we have selected to leverage the second oldest release available in our dataset (to which we refer as v_1). Although the oldest release in our dataset is v_0, we can calculate design churn only starting from v_1.
It is be possible that certain applications do not follow the trend observed across all applications combined together. This is why in the second setup (RQ1.2) we consider each application in isolation. For each application, the dataset contains all classes in one release (v_1). As opposed to the previous setup where the dataset is quite large, some applications feature only a limited number of Java classes. Therefore, for applications with less than 30 classes we have used the Wilcoxon test instead of the Welch's t-test and Fisher's exact test instead of the Chi-square test.

Prediction
Moving on, we are interested in finding out whether it is feasible to create a vulnerability prediction model based on design churn metrics. We leverage machine learning techniques in order to build a vulnerability prediction model.
Machine Learning Techniques. In general, to build a prediction model using machine learning, we select a number of Java classes to be part of the training set. These classes are characterized by their features (i.e., the metrics) and the corresponding classifications (as either vulnerable or clean). The training set is used to build the prediction model via a machine learning technique. We then apply the prediction model to a different set of Java classes, called the testing set. There are various machine learning techniques and their performance depends greatly on the characteristics of the data to be classified (Wolpert, 1997). Therefore, we have selected three different machine learning techniques to build vulnerability prediction models. All these techniques are widely used throughout the related work. Support Vector Machines (SVM) is a set of supervised learning methods that can be applied to classification (or regression) problems. Random Forests is an ensemble learning method for classification (and regression) that operate by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes output by individual trees. Naive Bayes is a classification technique based on Bayes' theorem and - despite its theoretical limitations - has been demonstrated to work quite well in many complex real-world situations. We present only the average results per application for each classification technique. Detailed results are available in their entirety online (KULeuven, 2013).

Performance Indicators. In order to assess the performance of a vulnerability prediction model we use precision, recall and their harmonic mean known as the F_1-score. These performance indicators are widely used throughout the related work and are largely accepted in the literature. The prediction model categorizes each Java class in the testing set as either vulnerable or clean. In order to calculate the performance indicators we compare the prediction with the observed value (as determined by the static code analyzer). There are four possible outcomes of the prediction, i.e. true positive (TP) if the class is both predicted and observed as vulnerable, true negative (TN) if the class is both predicted and observed as clean, false positive (FP) if the class is predicted as vulnerable but in fact is clean, and false negative (FN) if the class is predicted as clean but in fact is vulnerable. Precision is the percentage of all the vulnerable classes in the prediction and it is defined as:

$$P = TP/(TP + FP) \quad (1)$$

A vulnerability prediction model that has high precision would contain fewer false alarms and reduce the time wasted on validating files that are clean. Recall is the probability that a vulnerable class is classified as such and is defined as:

$$R = TP/(TP + FN) \quad (2)$$

A vulnerability prediction model that has high recall would reduce the risk of not validating a vulnerable file. The harmonic mean weighing both precision and recall equally, called the F_1-score, is defined as:

$$F_1 = 2*P*R/(P+R) \quad (3)$$

RQ2: Can we build a predictor using design churn metrics? In order to investigate RQ2 we have created two alternative experimental setups. Both experiments are based on building a prediction model that predicts whether Java classes in the subsequent (future) releases of the applications are vulnerable or clean.

Cross-project prediction model (RQ2.1). In the first experiment, we build one prediction model. The training set consists of all the classes in v_1 of all applications. The model is then tested on the subsequent releases of each application individually. For each application we calculate the average precision, recall and F_1-score values obtained over all releases.

Within-project prediction models (RQ2.2). In the second experiment, we build 10 prediction models, i.e., one for each application. For each prediction model, the training set consists of all classes in three releases (i.e., v_1, v_2 and v_3) of an application. We consider three releases in order to have a sufficiently large training set. Indeed, some applications have Java classes that do not change much from one release to another. Combined with the fact that some applications have only a limited number of Java classes, the machine learning techniques could be unable to learn anything meaningful due to the lack of sufficient training data. We test each model on the subsequent releases of the application (i.e., v_4 and following). Similar to the first experiment, we calculate the average precision, recall and F_1-score values obtained over all releases per application.

Selection of Applications

To validate our approach, we have selected the Android platform as it currently dominates the smart phone market (IDC, 2012; Zeman, 2011). As our study started in early 2012, we have selected applications in the time span of two years between the beginning of 2010 and the end of 2011. We selected the applications from the F-Droid repository of free and open source Android

applications. We have used three key criteria for the selection of the applications. First of all, the selected applications had to be open-source, as we need the source code to conduct our analysis. Given that this work focuses on the validation of design churn metrics, we have selected applications that had a minimum number of releases that we could download (at least 5 releases). In order to build reliable prediction models, we also required a certain minimum size (at least 1000 LOC). We have found a total of 10 applications that met these criteria out of over 200 applications available in the F-Droid repository. In Table 2, we present the ten applications we have selected for the purposes of our study.

Table 2
[Working set of applications]

Application	**Category**	**Downloads**	**Releases**
AnkiDroid	education	100k – 500k	5
BoardGameGeek	books & reference	10k – 50k	8
Connectbot	communication	1000k – 5000k	10
CoolReader	books & reference	1000k – 5000k	13
Crosswords	brain & puzzle	5k – 10k	16
FBReader	books & reference	1000k – 5000k	14
K9 Mail	communication	1000k – 5000k	19
KeePassDroid	tools	100k – 500k	13
MileageTracker	finance	100k – 500k	6
Mustard	social	10k – 50k	12

The applications are sorted alphabetically. The first column shows the names of the applications, the second column reports the application category (as appearing in the Google's app market), the third column provides an indication of the applications' popularity in terms of number of downloads and the last column reports the number of releases we have used in this study. All applications can be considered to be very popular. The applications are diverse in terms of type, ranging from social apps to games.

Figure 1 characterizes the releases of each application in terms of size as measured by their lines of code. Figure 2 presents the growth over time in terms of number of Java classes in the code base. The figures illustrate that there is diversity in the applications in terms of size. Further, all applications have grown throughout the releases. However, the growth follows quite different patterns. Note that the growth is not always monotonous, but some applications have actually shrunk throughout certain releases due to code refactoring (e.g. K9 Mail between releases 11 and 12).

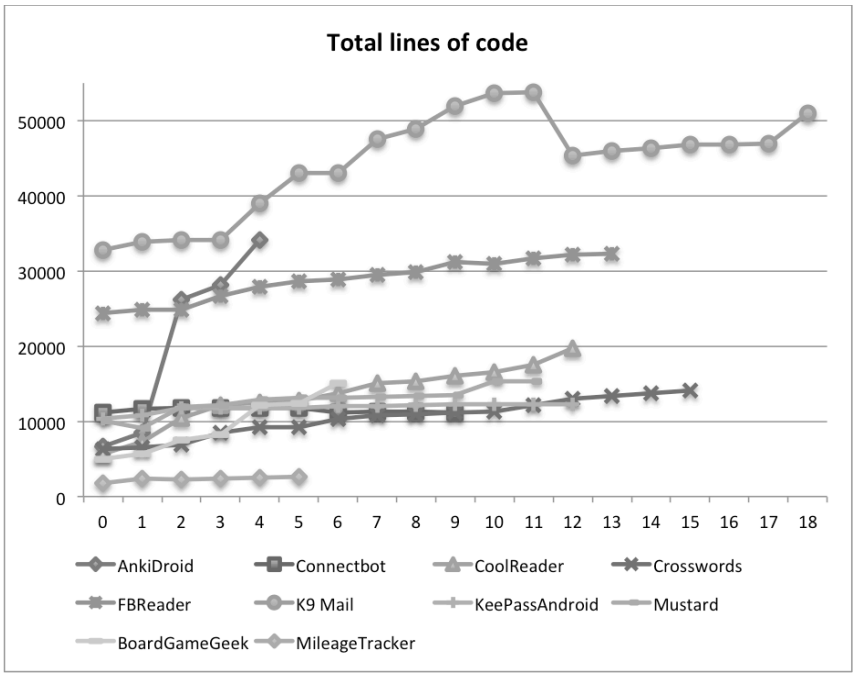

Figure 1. [Lines of code throughout the releases].

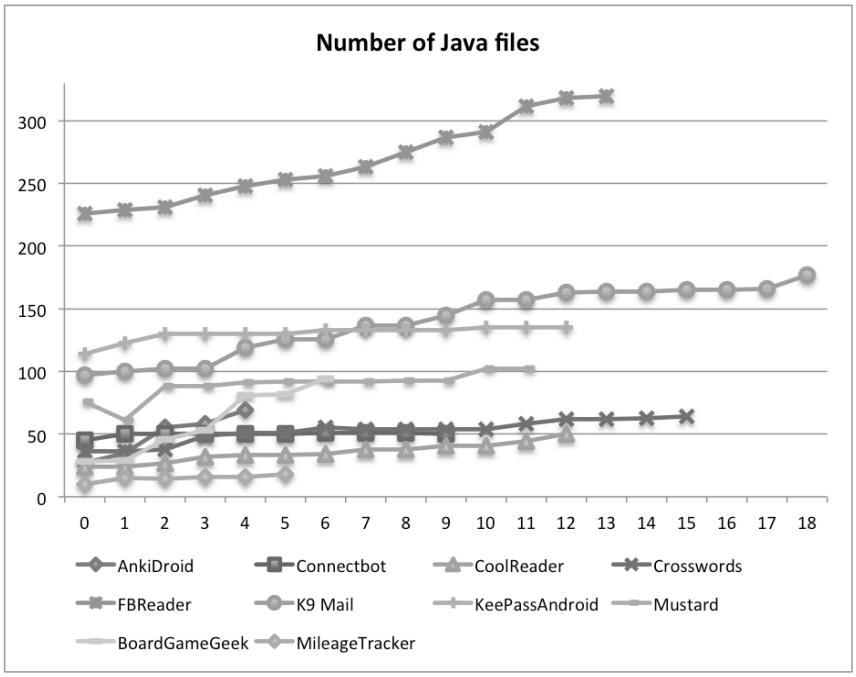

Figure 2. [Number of Java classes throughout the releases].

Figure 3 illustrates the results of Fortify analysis in terms of ratio of vulnerable classes across the releases for each application. Most applications have a slowly decreasing ratio of vulnerable classes. It seems that AnkiDroid has an erratic trend. CoolReader seems to be the only application with a deteriorating trend in terms of vulnerabilities across the releases.

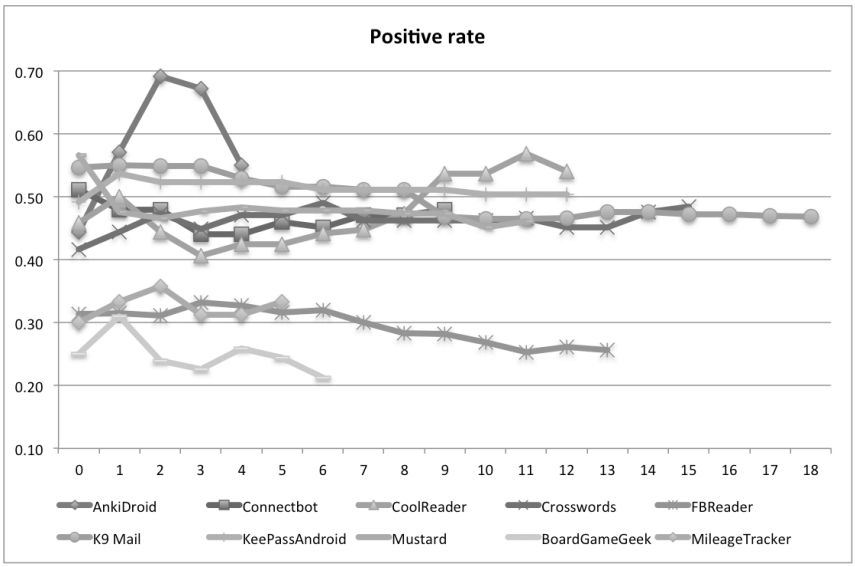

Figure 3. [Ratio of vulnerable classes throughout the releases].

Results

The next two subsections present (1) the results of the association between design churn metrics and the dependent variable and (2) the performance of the prediction models built on the design churn metrics.

Association

Association in All Applications (RQ1.1). Table 3 illustrates the results of the t-test statistical analysis that was performed on the combined dataset. The association in the last column indicates whether vulnerable classes have a higher value of the metric than the clean classes (+) or vice versa (-). Table 4 shows the results of the Chi-square test for independence. The relative risk coefficient represents the probability of a class being vulnerable when the NDBL metric is 1 over the probability of a class being vulnerable when the NDBL metric is 0 (i.e., ratio of conditional probabilities).

Table 3

[Association in all applications combined (interval metrics). V stands for a statistically significant difference in the mean with a p-value less than 0.05, X refers to the lack thereof, + indicates that the vulnerable classes have a higher mean, - refers to the reverse case]

Metric	**Difference**	**Association**
added fan-in	V	+
deleted fan-in	X	+
added fan-out	V	+
deleted fan-out	V	+

Table 4

[Association in all applications combined (nominal metrics)]

Metric	Not independent	RR	95% Confidence Interval
$NDBL_1$	V	1.636 (+)	[1.381–1.938]
$NDBL_2$	V	1.646 (+)	[1.383–1.959]
$NDBL_3$	V	1.558 (+)	[1.307–1.857]

As expected from intuition, NDBL and added/deleted fan-out are positively associated with vulnerabilities, and the association is statistically significant. It is quite surprising, however, that the one fan-in metric (added) is also significantly associated with vulnerabilities. A class should not be 'bothered' (from a security perspective) by other classes depending on it. Consequently, we expected vulnerabilities (similar to defects) to be related to the fan-out, not the fan-in of a class. A possible explanation is that added fan-in is a proxy for code churn that is demonstrated to be associated with security vulnerabilities (Shin, 2011). If more classes depend on a class, it is not unlikely that the class's implementation changed due to some new or changed requirements imposed by its new dependents.

Association in Single Applications (RQ1.2). We explored the associations in the context of each application in isolation. Table 5 shows the applications where the association is statistically significant. The association is always positive, i.e., vulnerable classes have higher mean value for the given metric.

Table 5
[Association in single applications]

Metric	Applications with positive association
added fan-in	CoolReader, Mustard
deleted fan-in	Mustard
added fan-out	CoolReader, BoardGameGeek, K9 Mail, KeePassDroid
deleted fan-out	CoolReader, BoardGameGeek, ConnectBot, K9 Mail, KeePassDroid
$NDBL_1$	BoardGameGeek, Crosswords, ConnectBot, K9 Mail
$NDBL_2$	BoardGameGeek, Crosswords, ConnectBot, K9 Mail
$NDBL_3$	BoardGameGeek, Crosswords, ConnectBot, K9 Mail

These results do not fully confirm the general trend depicted in Tables 3 and 4. That is, the association does not show up in all the applications. However, the root cause of this deviation is clear: some design churn metrics barely change between v_0 and v_1. Consider, for instance, Mustard that produces one of the most deviating behaviors, as fan-in churn is associated while all other metrics are not. Out of its 61 Java classes, only 11% have changed in terms of fan-out and NDBL metrics, while approximately 25% have changed in terms of fan-in. On the other hand, for K9 Mail, which does follow the general trend, 20% out of 99 classes have changed in terms of all design churn metrics.

For some applications (e.g., MileageTracker and FBReader), a substantial portion of the classes does change, but many of these classes are not vulnerable. Sometimes (although counter intuitively) implementations of difficult functionalities turn out to be less defective than small, swiftly implementable changes (Steff, 2012). In previous work, we attributed this effect to the developer attention, i.e., developers tend to pay more attention to difficult tasks. Thus, developers are not proportionately more likely to make mistakes in comparison to small tasks.

Vulnerability Prediction (RQ2)

Design churn is one aspect of software change. There are certainly others, e.g. evolution of code complexity, evolution of code size, and code churn. As a consequence, we do not expect design churn to cover all types of vulnerabilities. Therefore, we do not expect to have a high recall. We aim at a high precision, which would indicate that design churn is a valuable addition to the toolbox in terms of vulnerability prediction models.

Cross-Project Prediction Model (RQ2.1). In the first setup, we focus on the combined set of Java classes from all applications (w.r.t. their v_1 version). The combined dataset contains 702 Java classes, of which 289 are classified as vulnerable according to Fortify's SCA. Table 6 presents the summarized results of our findings, i.e., average precision, recall, F_1 values and p-rate per application.

Table 6

[Cross-project prediction]

	Random Forest			**Naïve Bayes**			**SVM**			
Application	**P**	**R**	**F_1**	**P**	**R**	**F_1**	**P**	**R**	**F_1**	**p-rate**
AnkiDroid	0.86	0.54	0.66	0.85	0.42	0.55	0.81	0.58	0.67	0.63
BoardGameGeek	0.45	0.31	0.35	0.36	0.17	0.22	0.46	0.37	0.40	0.24
ConnectBot	0.93	0.08	0.15	0.00	0.00	0.00	0.71	0.10	0.17	0.41
CoolReader	0.84	0.42	0.53	0.93	0.33	0.45	0.84	0.42	0.57	0.48
Crosswords	0.81	0.30	0.42	0.77	0.16	0.26	0.81	0.30	0.51	0.47
FBReader	0.45	0.14	0.20	0.65	0.07	0.12	0.45	0.14	0.26	0.29
K9 Mail	0.74	0.18	0.27	0.85	0.12	0.19	0.74	0.18	0.32	0.49
KeePassDroid	0.55	0.03	0.05	0.88	0.02	0.04	0.55	0.03	0.14	0.39
MileageTracker	0.22	0.24	0.21	0.46	0.14	0.21	0.22	0.24	0.27	0.33
Mustard	0.71	0.11	0.18	0.72	0.06	0.11	0.71	0.11	0.26	0.47
Overall average	**0.69**	**0.22**	**0.29**	**0.75**	**0.15**	**0.23**	**0.71**	**0.27**	**0.35**	

Based on the F_1-score, which is the harmonic mean of precision and recall, SVM machine learning technique produces slightly better performance indicators. However, the differences are small.

As expected, we can see the differences between the applications reflected in the precision and recall values. Recall is generally low. Precision is medium-to-high (above 70%) in at least half of the cases. Except for ConnectBot using Naïve Bayes technique precision values for all applications are well above the average positive rate (share of classes that are vulnerable), which represents the baseline. Indeed, a naive approach predicting all classes as vulnerable would achieve a precision equal to the p-rate. Note that the p-rate refers to the testing sets only.

The particular nature of MileageTracker and FBReader, as outlined above, plays a role in this experimental setup as well; with average precision and recall way below the overall average. BoardGameGeek and KeePassDroid also have rather low values. In the case of BoardGameGeek, we observed a complete refactoring in the third release, which we believe is responsible for the weak results. KeePassDroid, on the other hand, grows very slowly and barely has any churn that makes it hard to predict by definition.

Out of 95 releases the model predicted all classes as clean in 7 releases for Random Forest, 22 releases for Naïve Bayes and 3 releases for SVM respectively. Hence, for these special cases we

cannot calculate precision values, as it would involve a division by zero. We have determined that in these cases, applications barely changed between releases and kept exhibiting the same vulnerabilities in both releases. The change-based prediction model by definition cannot work under these conditions. If not enough change is present; other prediction models might be more appropriate.

Within-Project Prediction Models (RQ2.2). Although the generic model presented in the previous section is more likely to perform well across different sorts of Android applications, its granularity might be too coarse. Not all applications change in the same way, e.g., design churn metrics for AnkiDroid, which grows exponentially, are much higher in absolute numbers than those for KeePassDroid. Hence, in theory a prediction model crafted specifically for each application should result in improved numbers. Table 7 presents the average precision, recall, F_1 values and positive rate values of the prediction models created for each application (from three releases). Note that the p-rate refers to the testing sets only and the values are different from Table 6, as there are fewer releases to be tested.

Table 7
[Within-project prediction]

	Random Forest			**Naïve Bayes**			**SVM**			
Application	**P**	**R**	**F_1**	**P**	**R**	**F_1**	**P**	**R**	**F_1**	**p-rate**
AnkiDroid	0.51	0.86	0.64	0.92	0.32	0.48	0.54	1.00	0.70	0.54
BoardGameGeek	0.39	0.25	0.28	0.35	0.44	0.31	0.39	0.17	0.22	0.24
ConnectBot	0.77	0.52	0.61	0.83	0.12	0.21	0.77	0.55	0.75	0.41
CoolReader	0.95	0.33	0.46	0.98	0.20	0.31	0.92	0.33	0.48	0.49
Crosswords	0.80	0.40	0.48	0.83	0.32	0.45	0.79	0.44	0.52	0.47
FBReader	0.40	0.08	0.13	0.48	0.09	0.14	0.44	0.06	0.11	0.29
K9 Mail	0.62	0.57	0.59	0.82	0.18	0.27	0.64	0.61	0.62	0.49
KeePassDroid	0.44	0.04	0.06	0.66	0.05	0.09	0.63	0.01	0.03	0.39
MileageTracker	0.37	0.48	0.38	0.75	0.18	0.29	0.35	0.38	0.34	0.32
Mustard	0.63	0.52	0.56	0.47	0.97	0.63	0.64	0.54	0.57	0.47
Overall average	**0.64**	**0.38**	**0.43**	**0.71**	**0.28**	**0.31**	**0.66**	**0.40**	**0.45**	

The within-project prediction models considerably improve recall, as we expected. On the downside, however, they have slightly worse average precision. We believe that the main reason behind this behavior is that the cross-project prediction model has by definition much stricter criteria for classifying a class as vulnerable. Nevertheless, all applications except for AnkiDroid scored well above the p-rate. In the case of AnkiDroid, almost all classes were classified as vulnerable by Random Forest and SVM techniques, which virtually renders the prediction model useless. The within-project prediction models resulted in fewer releases that could not be predicted, i.e. where all classes were predicted as clean. Out of 75 releases that was the case for 3 releases for the Random Forest technique, 5 releases for the Naïve Bayes technique and 5 releases for the SVM technique respectively. Furthermore, as opposed to the cross-project prediction model where these special cases were distributed across 6 different applications, the within-project prediction models were mainly concentrated around KeePassDroid application. As mentioned previously, this application features many versions with virtually no design churn.

Once again the SVM technique results in best average performance based on the average F_1-score.

Threats to Validity

Construct validity.

The main threat to the construct validity is the use of Fortify's Static Code Analyzer to identify the vulnerable and clean Java classes. More specifically, it has been shown that SCA can produce a large number of false positives (Austin, 2011). To the best of our knowledge, there are no public databases containing a sufficient number of Android vulnerabilities. Hence, the use of vulnerability databases as the "ground truth" is not an obvious option. Furthermore, a number of recent publications have empirically verified that the warnings generated by SCA are in fact a good indication of NVD vulnerabilities (Walden, 2012; Edwards, 2012). Finally, similar correlations have also been reported in the area of software defect prediction (Nagappan, 2005).

Internal validity.

We have only considered the Java files and we have discarded the XML Android manifest packaged with each application. The manifest contains important security relevant information, such as the set of permissions that are required by the application.
We have considered a class as vulnerable if there was at least one vulnerability warning identified by Fortify. We neither considered the severity score (on five levels) reported by Fortify, nor did we take into account the number of vulnerability warnings per class. The results might not apply if only the top priority vulnerabilities are considered. Moreover, identifying a vulnerable class that contains many vulnerabilities might be more important than missing one with a single vulnerability. Further investigation is necessary in this respect.

External validity.

The number and the selection of applications allow us to reasonably generalize the results. Nonetheless, additional confirmatory studies using a different set of Android applications are necessary in order to generalize the achieved results to the entire class of Android applications.

Related Work

This section presents an overview of the related work in two categories, i.e., defect prediction and vulnerability prediction.

Defect Prediction

The work in the area of defect prediction is very broad and it is not our intention to be exhaustive. For a comprehensive survey on defect prediction we refer to the work of Catal (2009). We focus on works that employ measures, such as dependency structure and their change as those are closer to our study.
Kitchenham (1990) first used structural measures and found that the fan-out of code entities, i.e. the number of dependencies to other code entities, was useful in fault prediction. Schröter (2006) built prediction models for Eclipse plugins showing that dependencies on certain classes and components were strong predictors of defects. Zimmermann (2008) went one step further and

showed that local and global network measures such as density (local) and centrality (global) predict defects with a higher recall and comparable precision. They also used network measures as an aid for developers in indicating important files and binaries.
In our previous work (Steff, 2011), we added to this area of research by showing that the change of dependencies, i.e. structural change, was an even stronger indicator of faulty classes than static measures thus far presented in the literature. We introduced a graph kernel (Hido, 2009) to measure structural change between releases of software systems. One property of this kernel is that it allows the tracing of class-level changes and changes in classes' neighborhoods. We were able to show that not only change of a class's own dependencies, but also of its neighbors' dependencies is valuable for fault estimation. However, the number of dependencies changed seems to be less relevant than the mere fact that at least one dependency changed. We attribute this to the typical network properties of software dependencies. We have also investigated whether structural change has an added value over code churn (i.e. the number of modified lines of code of a class between versions) in defect prediction. We have determined that structural change and code churn indicate different defects, as some classes mainly suffer from defects incurred by structural changes and others mainly by code churn without simultaneous structural change. This observation has lead us to believe in the value of structural change for software vulnerability prediction, as an addition to commonly used traditional code complexity and code churn metrics.

Vulnerability Prediction
As opposed to defect prediction the research on security vulnerability prediction is more limited. This section presents a representative overview of works on vulnerability prediction that focus on categorizing software components into either "vulnerable" or "clean".
Neuhaus (2007) have investigated the correlation between vulnerabilities and include statements. The proposed approach was validated in the context of the Mozilla project with the reported average precision of 0.70 and recall of 0.45. Neuhaus & Zimmermann (2009) have demonstrated that software vulnerabilities correlated with dependencies between packages. The authors have leveraged the support vector machines (SVM) technique in order to build prediction models based on dependencies. The models have identified vulnerable packages in the context of 3241 Red Hat packages with a median precision of 0.83 and a median recall of 0.65. Zimmermann (2010) found a correlation, albeit weak, between vulnerabilities and various software and development metrics. Nguyen & Tran (2010) proposed an approach to predict vulnerable components using component dependency graphs. The approach was validated on two versions of Mozilla Firefox JavaScript Engine (JSE). For JSE version 1.5 the authors have reported an average precision and recall of 0.6 and 0.68 respectively. For JSE version 2.0 the reported performance indicators were 0.6 and 0.61 for precision and recall respectively. Shin, Meneely et al (2011) have built a vulnerability prediction model based on complexity metrics that was able to predict security problems in Mozilla JavaScript Engine with an average recall of 0.80 (precision was not reported, but was mentioned to be very low). Chowdhury et al (2011) have determined that complexity, coupling and cohesion metrics were able to predict majority of the vulnerability-prone files in Mozilla Firefox (0.73 accuracy and 0.74 recall), with tolerable false positive rates (0.29). Finally, two research initiatives have focused on the question whether it makes sense to create specialized vulnerability prediction models given the abundance of fault prediction models that have been demonstrated to be effective. Shin & Williams (2011) conclude that fault prediction models based upon traditional metrics can substitute for specialized

vulnerability prediction models. Unfortunately, both fault prediction and vulnerability prediction models produce relatively high false positive rate. Gegick et al (2009) reach a similar conclusion. Smith et al (2011) investigated the predictive power of the places containing a large number of SQL statements, i.e., SQL hotspots. The authors determined that the more SQL hotspots a file contains per lines of code, the higher the probability that the file will contain any type of vulnerability. This vulnerability prediction model scores between 0.02 and 0.5 for precision and 0.1 and 0.4 for recall for the WordPress blog engine, and between 0.04 and 1.0 for precision and 0.09 and 1.0 for recall for the WikkaWiki application.

Conclusions

In this paper, we explored the use of several design churn metrics in vulnerability prediction. For validation, we selected ten Android applications. First, we investigated which metrics are associated to vulnerabilities in each application. While design churn is significantly associated to vulnerabilities across the set of all applications, individual applications may not show a significant association. Therefore, design churn is a tool to be selectively applied where its utility has been ascertained. Second, we have evaluated the use of design churn as a predictor of vulnerabilities. We observed a relatively high precision, but low recall.

The results of this study motivate future work in various areas. First, the prediction performance could be improved if churn metrics were complemented by other metrics, such as size, coupling and cohesion. Second, we also plan on validating the proposed approach on a different set of applications, including those featuring a large set of documented vulnerabilities in existing databases such as the National Vulnerability Database.

References

Austin, A., & Williams, L. (2011). One technique is not enough: A comparison of vulnerability discovery techniques. In, *International Symposium on Empirical Software Engineering and Measurement (ESEM)*.

Basili, V.R., Briand, L.C., & Melo, W.L. (1996). A Validation of Object- Oriented Design Metrics as Quality Indicators. *IEEE Transactions on Software Engineering*, 22(10), 751-761.

Catal, C., & Diri, B. (2009). A systematic review of software fault prediction studies. *Expert Systems with Applications*, 36(4), 7346–7354.

Chowdhury, I., & Zulkernine, M. (2011). Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal System Architecture* 57(3), 294–313.

Dagenais, B., & Hendren, L. (2008). Enabling static analysis for partial java programs. *ACM Sigplan Notices*, 43(10), 313-328.

Edwards, N., & Chen, L. (2012). A historical examination of open source releases and their vulnerabilities. In, *Proceedings of the 2012 ACM conference on Computer and communications security* (pp. 183–194). ACM, New York, NY, USA.

Gegick, M., Rotella, P., & Williams, L. (2009). Toward non-security failures as a predictor of security faults and failures. In, *Proceedings of the 1st International Symposium on Engineering Secure Software and Systems* (pp. 135–149). Springer-Verlag, Berlin, Heidelberg.

Gyimothy, T., Ferenc, R., & Siket, I. (2005). Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10), 897–910.

Hido, S., & Kashima, H. (2009). A linear-time graph kernel. In, *IEEE International Conference on Data Mining* (pp. 179–188).

IDC: Android and iOS surge to new smartphone OS record in second quarter of 2012. Retrieved September 1, 2012, from http://www.idc.com/getdoc.jsp?containerId=prUS23638712

Kitchenham, B., Pickard, L., & Linkman, S. (1990). An evaluation of some design metrics. *Software Engineering Journal*, 5(1), 50–58.

KU Leuven (2013). Experiment materials. http://goo.gl/asB5H

McGraw, G., (2006). *Software Security: Building Security*. Addison-Wesley Professional.

Menzies, T., Greenwald, J., & Frank, A. (2007). Data Mining Static Code Attributes to Learn Defect Predictors. *IEEE Transactions on Software Engineering*, 33(1), pp. 2-13.

Nagappan, N., & Ball, T. (2005). Static analysis tools as early indicators of pre-release defect density. In, *International conference on Software Engineering (ICSE)*, (pp. 580-586).

Neuhaus, S., & Zimmermann, T. (2009). The beauty and the beast: vulnerabilities in red hat's packages. In, *Proceedings of the 2009 conference on USENIX Annual technical conference*, (pp. 30–30). USENIX Association, Berkeley, CA, USA.

Neuhaus, S., Zimmermann, T., Holler, C., & Zeller, A. (2007). Predicting vulnerable software components. In, *ACM Conference on Computer and Communications Security (CCS)* (pp. 529-540). ACM, New York, NY, USA.

Nguyen, V.H., & Tran, L.M.S. (2010). Predicting vulnerable software components with dependency graphs. In, *Proceedings of the 6th International Workshop on Security Measurements and Metrics*, (pp. 3:1–3:8). ACM, New York, NY, USA.

Schröter, A., Zimmermann, T., & Zeller, A. (2006). Predicting component failures at design time. In, *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, (pp. 18–27). ACM, New York, NY, USA.

Shin, Y., Meneely, A., Williams, L., & Osborne, J.A. (2011). Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Transactions on Software Engineering*, 37(6), 772–787.

Shin, Y., & Williams, L. (2011). Can traditional fault prediction models be used for vulnerability prediction? *Empirical Software Engineering*, 18(1), 1–35.

Smith, B. & Williams, L. (2011). Using SQL hotspots in a prioritization heuristic for detecting all types of web application vulnerabilities. In, *IEEE International Conference on Software Testing, Verification and Validation (ICST)*.

Steff, M., & Russo, B. (2011). Measuring architectural change for defect estimation and localization. In, *International symposium on Empirical Software Engineering and Measurement* (pp. 225–234).

Steff, M., & Russo, B. (2012). Characterizing the roles of classes and their fault-proneness through change metrics. In, *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement* (pp. 59–68).

Walden, J., & Doyle, M. (2012). SAVI: Static-analysis vulnerability indicator. *IEEE Security & Privacy* 10(3), 32–39.

Wolpert, D., & Macready, W. (1997). No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1), 67–82.

Zeman, E. (2011). Android, IOS crush blackberry market share. Retrieved September 2012, from http://www.informationweek.com/mobile/mobile-devices/android-ios-crush-blackberry-market-share/d/d-id/1104538?

Zimmermann, T., & Nagappan, N. (2008). Predicting defects using network analysis on dependency graphs. In, *International Conference on Software Engineering* (pp. 531–540).

Zimmermann, T., Nagappan, N., & Williams, L. (2010). Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In, *International Conference on Software Testing, Verification and Validation (ICST)*.